ARMY RESEARCH LABORATORY

The Implementation of a Python Class for Structuring Network Data Collected in a Test Bed

by Binh Q. Nguyen

ARL-TR-4423 **April 2008**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Adelphi, MD 20783-1197

ARL-TR-4423 **April 2008**

The Implementation of a Python Class for Structuring Network Data Collected in a Test Bed

Binh Q. Nguyen
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
April 2008	Final	January to February 2008

4. TITLE AND SUBTITLE
The Implementation of a Python Class for Structuring Network Data Collected in a Test Bed

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Binh Q. Nguyen

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRD-ARL-CI-CN
2800 Powder Mill Road
Adelphi, MD 20783-1128

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-4423

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report documents an internally developed Python class that takes in a set of data files and organizes them into effective data structures that are suitable for the subsequent extraction, processing, and analysis. The report includes usage examples by describing Python snippets that perform statistical calculations and that transform the data into comma-separated values. Sample input and output data are appended to the report.

15. SUBJECT TERMS
Python, IDS data, network utilization, network performance

16. Security Classification of:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Binh Q. Nguyen
U	U	U	U	30	19b. TELEPHONE NUMBER (*Include area code*) (301) 394-1781

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

Intentionally Left Blank

1. Introduction

The U.S. Army Research Laboratory (ARL) has been developing intrusion detection systems (IDS) intended for tactical wireless mobile ad-hoc network (MANET) systems through various research programs, e.g., the Collaborative Technology Alliance (CTA) and Science and Technology Objective (STO) programs. ARL also empirically evaluates newly developed and experimental IDS in its wireless emulation laboratory (WEL).

In a recent evaluation, ARL captured a set of data describing quantitative network utilization and performance and saved them in a *record* file for each run. (The description and an example of a *record* file are presented respectively in subsequent paragraphs and in appendix A.) To parse these *record* files, ARL had a computer program[1] implemented as a Python[2] script containing two main functions: parseAllFiles and separateTestsByKeys. The former was to read, parse, and return the data stored in a list of dictionaries, and the latter was to take the returned dictionary to create another dictionary whose keys are the concatenation of three types of data: (1) attack type, (2) node density, and (3) detector type. The functional behavior of the former appeared to perform as expected, but that of the latter was still to be tested and evaluated for correctness and usability.

As a script, the computer program was written in a procedural-programming style providing a specific functionality, solving an exact problem at a certain time. The script thus was designed to run as an independent Python application, not as a module that could be imported in a Python shell for an interactive session or incorporated into another module. Therefore, as an alternative to the script, a new Python class was designed and implemented, and it is called the ParseIDSdata class.

This report documents the ParseIDSdata class that also takes in a set of *record* files and organizes them into a list of dictionaries as the current script does. That is the only common functionality between the current script and the new class. The ParseIDSdata class additionally provides the user a set of data services, implemented as callable methods, for retrieving specific types of data in the input files. The retrieved data are then used for subsequent analysis and processing. As the term "class" implies, it is built using object-oriented (OO) techniques that offer the benefits of reusability, extensibility, and maintainability. In addition, because the class was implemented using Python, the user can import the class into a Python shell for an interactive session with the class, experimenting with its features or developing new applications.

The intended readers of this report include technical personnel who are interested in using the ParseIDSdata class to inspect, extract, or process *record* files. The report includes three usage

[1]The functions were created by Cem Karan of ARL in 2007.

[2]The Python computer programming language, http://www.python.org.

examples of the ParseIDSdata class. The first example presents a screenshot of an interactive session. The second example provides an implemented method that extracts a specific set of data and calculates its statistics. The last example illustrates a way to transform a set of data into comma-separated values. The internal structures, operations, and interfaces of the examples are explained in this report.

The rest of this report consists of four sections. The next section describes the structure of a *record* file. Section 3 presents and explains the functional behavior and the outputs of the ParseIDSdata class. Section 4 gives usage examples of the ParseIDSdata class. Section 5 concludes the report.

---

2. The *Record* File

Each record file consists of two sections: (1) a metadata section and (2) a data section. The metadata section describes the informational contents in the file. The data section has one or more named blocks of data. Each block consists of data lines. Each data line is a name-value pair (e.g., minRxHsls = 96). Non-data lines, such as comments and blank lines, are also included in the file to improve readability and intelligibility. An input line is a comment line if its first visible character is the percent (%) character. The structure of the file is depicted below:

```
Metadata { Data Block 1 } { Data Block 2 } ... { Data Block N }
```

2.1 Metadata

The metadata section has at most five name-value pairs to provide information about the following data blocks. The five names include NODE_DENSITY, DATE & TIME, Attack Type, Attacker Node, and DETECTOR. The first four names always appear in the sample set of data, but the last one, the DETECTOR name, does not consistently appear.

```
DATE & TIME=        08-22-07_14:40
NODE_DENSITY=       low_dense
Attacker Node=      node47
Attack Type=        FAST HELLO
DETECTOR=           SPEC
```

The value of the DATE & TIME field is unique in every data file, and it matches the name suffix of an input file, e.g., record_08-22-07_14:40. (On Microsoft Windows, the ':' character cannot be used to name a file. Therefore, it is converted into the '_' character.) The values of other

fields are not unique. For example, the value of the NODE_DENSITY field of many input data files have the same value, e.g., low_dense .

2.2 Data Blocks

Each data block is enclosed between a pair of curly brackets "{}". The open-curly bracket '{' signifies the start of a data block, and the close-curly bracket '}' signifies the end. The brackets themselves are not part of the data. Each data block has a set of keywords describing its own contents. The "Name" key word is a special one because it is the name of the associated block. Below is a short example of a data block:

```
{
        Name = PreBANDWIDTH COMSUMPTION
        %All Bandwidth Consumption Calculations are in bps
AllReceivedTraffic = 434.41
...
TransmittedPmsTraffic = 245.80
}
{
                Name = BANDWIDTH COMSUMPTION
                %All Bandwidth Consumption Calculations are in bps

AllReceivedTraffic =  1.3857e+05
...
minTxPms = 482.40
}
{
                Name = CALCULATIONS LATENCY
Attack-Jmap =                   31
...
False Detection =               0
}
```

3. The ParseIDSdata Class

The ParseIDSdata class is designed to read in the contents of *record* files and organize them in data structures for subsequent extraction and processing. Invoking the class requires (1) the specification of the directory in which the data files reside (*e.g., records*) and (2) the common name prefix of all the data files (*e.g., record_*). The following statement instantiates an object of type ParseIDSdata:

```
o = ParseIDSdata(datdir="records", prefix="report_")
```

3.1 Interfacing Methods

The class provides several methods for retrieving and printing a set of specific data. The data retrievers always return to their caller an object referring to the requested data. The data printers format and print the requested data to the screen. The names of the methods are listed and explained below.

Data Retriever	Data Printer
`get_metadata_list()`	`print_metadata()`
`get_name_set()`	`print_name_set()`
`get_idsinfo()`	`print_idsinfo()`
`get_attack_types()`	`print_attack_types()`
`get_datalist()`	
`extract_data()`	

- The get_metadata_list method returns a list of dictionaries. The length of the list tells the number of input data files because each file has a metadata section. Each dictionary stores the metadata of each file. The print_metadata method prints the metadata of each input file to the screen as shown below:

```
NODE_DENSITY = low_dense
 ATE & TIME = 08-17-07_14:20
Attacker Node = node41
 ttack Type = DROP VALID NEIGHBOR
…
…
NODE_DENSITY = low_dense
DATE & TIME = 08-17-07_14:08
Attacker Node = node27
Attack Type = DROP VALID NEIGHBOR
```

- The get_name_set method returns the names of all the data blocks in a set of strings. The size of the set indicates the number of distinct categories of input data. Each string is the name of a data block or that of the metadata section. The print_name_set method prints the names to the screen. The example below shows four different categories of data:

```
Block Name: BANDWIDTH COMSUMPTION
Block Name: CALCULATIONS LATENCY
Block Name: PreBANDWIDTH COMSUMPTION
Block Name: Metadata
```

- The get_idsinfo method returns the information about all the input data in a dictionary. The keys of the dictionary are derived from the name-value pairs found in the metadata section of all the input files. The value of each key is a set of a particular type of data, which can be inspected by printing them on the screen using the print_idsinfo method. The output of the print_idsinfo method reveals information about the input data. For example, two types of detectors (SPEC and SPARTA) were found in the input data as shown below:

```
NODE_DENSITY  : set(['low_dense','medium_dense','high_dense'])
DATE & TIME : set(['08-17-07_14:20', ..., '08-21-07_14:32'])
DETECTOR : set(['SPEC', 'SPARTA'])
Attacker Node : set(['node27', ...,  'node41'])
Attack Type : set(['FAST HELLO','DROP VALID NEIGHBOR'])
```

The length of each set tells the number of distinct categories in the set. The length of the NODE_DENSITY set indicates the number of node densities. The length of the DATE & TIME set shows the number of data files because every file has this information. The length of the Attacker Node set reveals the number of nodes that act as attackers. The length of the Attack Type set reports the number of attack types in the data files.

- The get_attack_types method returns a list of data associated with each type of attack in a dictionary. Each key of dictionary is the name of an attack, and its value is a list of data having the same attack type. The print_attack_types method the name of each attack and the number of files that are associated with the attack; it does not print the data because of their voluminous nature. An example of its output is shown below:

```
AttackType (PIM SEND BAD REGISTER): list(dict[k]) (28 items)
AttackType (DROP VALID NEIGHBOR): list(dict[k]) (30 items)
AttackType (PIM DROP PRUNE/JOIN): list(dict[k]) (30 items)
AttackType (Bad Local Neighbor): list(dict[k]) (67 items)
AttackType (FAST HELLO): list(dict[k]) (30 items)
AttackType (PIM BAD NEXT HOP): list(dict[k]) (30 items)
```

By inspecting the output of the print_attack_types method, one can find that the input data set has 30 files having the same `PIM BAD NEXT HOP` attack type. To retrieve the data of these 30 files and to print out the contents of the first file, the following code is provided:

```
o = ParseIDSdata()
data_list = o.get_attack_types().get('PIM BAD NEXT HOP')
print_dict( data_list[0] )
```

- The get_datalist method returns an object referring to the entire structured contents of all the input files that are organized in a list of dictionaries; that is, each element of the list is a dictionary containing the data of each file. The keys of the dictionary are the names of the data blocks in an input file. The value of each key is the associated block of data which is also organized as a dictionary storing the name-value pairs of data. The following Python code illustrates the data structure by printing out all the data in the structure.

```
for input_data  in get_datalist():
    for blockname, blockdata in input_data.iteritems():
        print 'Data Block Name:', blockname
        for field, value in blockdata.iteritems():
            print '%15s = %s' % (field, value)
```

- The extract_data method extracts and returns a list of data according to a specified set of criteria: (1) attack type, (2) node-density type, and (3) detector type. If nothing meets all of the criteria, then it returns an empty list. Each element of the list is a dictionary. They keys of the dictionary are the names of the data blocks, and the values of the dictionary are also dictionaries, which are structures that store name-value pairs of each data block. The following Python code uses the extract_data method to extract a set of data that meet the given criteria and prints out the metadata information of the data.

```
attack = 'Bad Local Neighbor'
dense  = 'low_dense'
detect = 'SPARTA'
o = ParseIDSdata()
for data_dict in o.extract_data(attack, dense, detect):
    print_dict( data_dict.get(KEY_META) )
```

Output:

```
NODE_DENSITY  : low_dense
DATE & TIME : 08-27-07_17:31
DETECTOR : SPARTA
Attacker Node : node41
Attack Type : Bad Local Neighbor
...
...
NODE_DENSITY : low_dense
DATE & TIME : 08-27-07_18:00
DETECTOR : SPARTA
Attacker Node : node27
Attack Type : Bad Local Neighbor
```

The following Python code uses the extract_data method to extract a set of data that meet every possible criteria and prints out their associated file names.

```
o             = ParseIDSdata()
idsinfo       = o.get_idsinfo()
attack_types  = idsinfo.get(KEY_ATTACK_TYPE)
density_types = idsinfo.get(KEY_NODE_DENSITY)
detector_types= idsinfo.get(KEY_DETECTOR)
detector_types.add(None)
for attack in attack_types:
  for density in density_types:
    for detect in detector_types:
      data_list = o.extract_data(attack, density, detect)
      print '%s - %s - %s: (%d files)' % (
        attack, density, detect, len(data_list))
      for data_dict in data_list:
          print'\treport_%s'%
data_dict.get(KEY_META).get(KEY_DATE_TIME))
```

Output:

```
PIM SEND BAD REGISTER - high_dense - SPEC: (0 files)
PIM SEND BAD REGISTER - high_dense - SPARTA: (0 files)
DROP VALID NEIGHBOR - low_dense - None: (10 files)
        report_08-31-07_11:20
        report_08-31-07_13:57
          ...
        report_08-31-07_15:49
        report_08-31-07_16:05
DROP VALID NEIGHBOR - low_dense - SPEC: (0 files)
DROP VALID NEIGHBOR - low_dense - SPARTA: (0 files)
DROP VALID NEIGHBOR - medium_dense - None: (10 files)
        report_08-17-07_13:41
        report_08-17-07_13:54
          ...
        report_08-17-07_14:32
        report_08-17-07_14:44
```

Furthermore, the ParseIDSdata class also organizes the input file names according to the information in the metadata section and creates an index file containing hyperlinks to the data files. This feature is designed to provide the user a way to visually inspect an input data file. The index file can be viewed using a web browser. Appendix B shows a shorten version of an index file.

3.2 Internal Methods

This section describes and explains some internal methods in the ParseIDSdata class. Although the implementation of these methods is transparent to the user, its description is intended to assist the user in the understanding of the ParseIDSdata class, which will facilitate its improvement in the future. Another purpose is to highlight special features of the Python programming language that enable the expression of a solution in fewer lines of code. The special features are implemented in the following three functions: (1) the filter_data input-data reader, (2) the list2dict data converter, and (3) the print_dict recursive dictionary printer. Only the first two functions are part of the ParseIDSdata class.

- **Input Data Reader**. Given the name of an input data file to which the fn variable refers, the filter_data function performs six tasks: (1) opening the data file, (2) reading its contents line by line, (3) removing any extra spaces preceding and trailing a line of text, (4) eliminating blank lines and comments, (5) placing the filtered data in a Python data structure called list, and (6) returning it to the caller. If the function encounters a system error, it prints out an error message, e, and returns an empty list to the caller. Completing all these tasks requires only a single line of Python code as shown below:

```
def filter_data(fn):
  try:
    return [s for s in [ s.strip() for s in open(fn).readlines()]
            if len(s) and not s.startswith(COMMENT_CHAR) ]
  except IOError, e:
    print_error(e)
    return list()
```

- **Data Structure Converter.** Given a Python list of name-value pairs, s, the list2dict function performs four tasks: (1) splitting each name-value pair into two strings, (2) removing any extraneous white spaces surrounding the splitted strings, (3) creating a Python dictionary, and (4) returning it to the caller. Completing all these tasks requires only a single line of Python code as shown below:

```
def list2dict(s):
  return dict([(x[0].strip(), x[1].strip())
              for x in [x.split("=") for x in s ]])
#name=value --> [name, value] to which x refers
#           --> x[0] = name, x[1]=value
#           --> dict[name]=value
#           --> dict[x[0]]=x[1]
```

- **Dictionary Printer**. Given a Python dictionary to which the variable named d refers, the print_dict function recursively prints its key-value pairs on the screen. The function displays two formatted columns of name-value pairs: the name is on the left, and the value is on the right. The field length of the name is computed by taking the maximum value of the calculated length of each key in the dictionary. Doing so requires a single expression (`max([len(x) for x in d.keys()])`). The computed value is then incorporated in the output format as shown below:

```
def print_dict(d):
    fmt='%%%ds : %%s' % max([len(x) for x in d.keys()])
    for k, v in d.iteritems():
        if type(v) == type(dict()):
            print 'Key = <%s>' % k
            print_dict(v)
        else:
            print fmt % (k, v)
    print
```

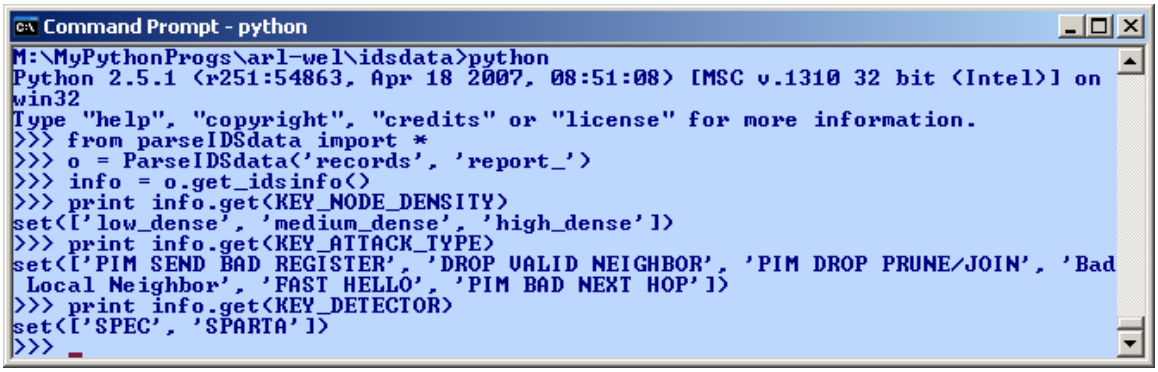
---

4. Usage Examples

This section provides three examples of Python snippets to illustrate different uses of the ParseIDSdata class. Example 1 shows a screenshot of an interactive session. Example 2 explains the code that was used to extract the values of a specific field name. Example 3 presents the code that extracts a subset of the IDS data and converts them into comma-separated values.

4.1 Example 1

This example provides snippets of Python code that uses the basic data services provided by the ParseIDSdata class to retrieve three data categories in the test data files: (1) node density, (2) attack type, and (3) detector type. The code uses the ParseIDSdata class to read in all the data files having their names beginning with "report_" in the "records" directory, which is located at the current directory (M:\MyPythonProgs\arl-wel\idsdata). The variable "o" refers to the object of type ParseIDSdata which has been instantiated successfully. The next statement, info = o.get_idsinfo(), retrieves the IDS information to which the variable "info" refers. The information is organized in a dictionary whose keys were derived from the name-value pairs in the metadata section.



4.2 Example 2

This example shows the code necessary for computing a basic set of statistics of the values of the Attack-Jmap data field that must meet a set of two criteria: (1) the node density must be of type low_dense and (2) the attack type must be of type PIM DROP PRUNE/JOIN. Because the specified Attack-Jmap data field exists only in the CALCULATIONS LATENCY block, the whole block data must be first extracted, and then the value of the target field is extracted from the data block.

```
from parseIDSdata import *
DATDIR      = 'records'
PREFIX      = 'report_'
BLOCK_NAME  = 'CALCULATIONS LATENCY'
ATTACK_TYPE = 'PIM DROP PRUNE/JOIN'
DENSE_TYPE  = 'low_dense'
FIELD_NAME  = 'Attack-Jmap'

o           = ParseIDSdata(DATDIR, PREFIX)
results     = list()
for input_data  in o.get_datalist():
    meta_data   = input_data.get(KEY_META)
```

```
    #print_dict(meta_data)
    if meta_data.get(KEY_NODE_DENSITY) != DENSE_TYPE:
           continue
    if meta_data.get(KEY_ATTACK_TYPE) != ATTACK_TYPE:
           continue
    block_data  = input_data.get(BLOCK_NAME)
    if FIELD_NAME in block_data:
        field_value = block_data.get(FIELD_NAME)
        results.append(float(field_value))
        print '%s = <%s>' % (FIELD_NAME, field_value)

print '-----'
print 'unsorted =', results
results.sort()
print 'sorted   =', results
print '-----'
print 'max=%.3f, min=%.3f, ave=%.3f' % (results[-1],
              results[0], sum(results)/len(results))
```

Output:

Attack-Jmap = <149>
Attack-Jmap = <128>
...
Attack-Jmap = <149>
Attack-Jmap = <93>
-
unsorted = [149.0, 149.0, 148.0, ..., 149.0, 148.0, 93.0]
sorted   = [93.0, 128.0, 140.0, ..., 149.0, 149.0, 149.0]
-
max=149.000, min=93.000, ave=140.100

4.3 Example 3

This example provides a way to extract bandwidth-consumption data in "high_dense" (high node density) scenarios and save them in a comma-separated file, which is ready to be used by a spreadsheet program or by a database management system capable of importing comma-separated values. The code consists of two main functions: (1) retrieve the specified data, and (2) convert the data into comma-separated values and save them to the idsdata.csv file.

```
BW = 'BANDWIDTH COMSUMPTION'
TARGET_DENSITY  = 'high_dense'

def retrieve_data():
    results     = list()
```

```
    for data in ParseIDSdata().get_datalist():
        metadata = data.get('Metadata')
        if metadata.get('NODE_DENSITY') == TARGET_DENSITY:
            results.append(data.get(BW))
    return results

def save_as_csv_file(datalist, fn):
    titles = set()
    for data in datalist:
        titles.update(data.keys())
    titles =  sorted(titles)

    ofd = open(fn, 'w')
    for title in titles:
        print >> ofd, '%s,' % (title),
    print >> ofd

    for data in datalist:
        for title in titles:
            print >> ofd, '%s,' % (data.get(title)),
        print >> ofd
    ofd.close()

#-------------------------------------

data = retrieve_data()
save_as_csv_file(data, fn='idsdata.csv')
```

Output File Name: 'idsdata.csv'

File Contents:
AllRecTraffic, AllTransTraffic, ... minTxPms, minTxTraf,
3.2479e+05, 4.6541e+04, ... , 7832.8, 43.200,
...

5. Summary and Conclusion

The description in this report of the ProcessIDSdata class and its interfaces shows how the class is designed to read in a set of IDS data files and then organize their contents in effective data structures that are highly suitable for subsequent analysis and processing. The usage examples illustrate the built-in functions to extract a specific set of data, calculate their statistics, and view the results using a web browser. The third example illustrates the implementation of an algorithm for extracting a set of specific data and then converting them into a comma-separated file. The report also stresses some special features of the Python programming language that enable the expression of algorithmic solutions in fewer lines of code.

INTENTIONALLY LEFT BLANK

Appendix A. The Structure of *Record* Files

```
DATE & TIME= 11-01-07_11:38
NODE_DENSITY=          low_dense
Attacker Node=   node92
Attack Type=       PIM DROP PRUNE/JOIN

{
                    Name = PreBANDWIDTH COMSUMPTION
                    %All Bandwidth Consumption Calculations are in bps
AllReceivedTraffic =  2.9269e+05
maxRxTraf =  1.3423e+07
minRxTraf = 0
…
…
maxTxHsls = 288
TransmittedPmsTraffic =  3.0573e+05
maxTxPms =
13422568
minTxPms = 0
}
{
                    Name = BANDWIDTH COMSUMPTION

                    %All Bandwidth Consumption Calculations are in bps

AllReceivedTraffic =  1.7987e+05
maxRxTraf =  3.0944e+06
minRxTraf = 787.20
…
…
TransmittedPmsTraffic =  1.4553e+05
maxTxPms =  3.0707e+06
minTxPms = 563.20
}
{
                    Name = CALCULATIONS LATENCY

Attack-Jmap =                    93
Snort-Jmap =                     0
Jmap-Pms =                       1
PMS-SmartFirewall =              14
…
…
False Detection =                0
Detected =                       1
Miss Detection =          0
Initialization Time =            203
}
```

INTENTIONALLY LEFT BLANK

Appendix B. Index File

PIM BAD NEXT HOP: report_10-03-07_17_44 report_10-03-07_18_34 report_10-04-07_18_13 report_10-04-07_18_27 report_10-04-07_18_42
PIM SEND BAD REGISTER: report_09-16-07_19_36 report_09-16-07_19_50 report_09-17-07_10_21 report_09-17-07_10_32 report_09-17-07_19_34
DROP VALID NEIGHBOR: report_08-17-07_13_41 report_08-17-07_13_54 report_09-05-07_16_41 report_09-05-07_16_57 report_09-05-07_17_09
Sparta: report_08-27-07_17_31 report_08-27-07_17_47 report_08-27-07_18_00 report_08-27-07_18_11 report_08-28-07_18_07
node42: report_09-13-07_16_54 report_09-13-07_17_07 report_09-13-07_17_33 report_10-05-07_11_22 report_10-05-07_11_34 report_10-05-07_11_45
high_dense: report_08-31-07_16_38 report_08-31-07_16_49 report_08-31-07_17_00 report_08-31-07_17_11 report_08-31-07_17_22 report_08-31-07_17_33 report_10-30-07_13_12 report_10-30-07_13_24
SPEC: report_08-27-07_11_29 report_08-27-07_11_47 report_08-27-07_12_00 report_09-11-07_10_34 report_09-11-07_10_45 report_09-11-07_10_56 report_10-17-07_14_37 report_10-17-07_14_48
SPARTA: report_09-04-07_16_58 report_09-04-07_18_00 report_09-05-07_10_43 report_09-05-07_10_56 report_09-05-07_12_31 report_09-05-07_12_41
PIM DROP PRUNE/JOIN: report_09-13-07_16_54 report_09-13-07_17_07 report_09-13-07_17_33 report_09-13-07_17_47 report_09-13-07_17_57 report_09-13-07_18_08 report_11-01-07_11_38
Bad Local Neighbor: report_08-27-07_11_29 report_08-27-07_11_47 report_08-27-07_12_00 report_08-27-07_15_22 report_08-27-07_15_36 report_08-27-07_15_52 report_10-17-07_12_29 report_10-17-07_14_26 report_10-17-07_14_37
low_dense: report_08-21-07_14_32 report_08-21-07_17_39 report_08-21-07_17_51 report_08-22-07_10_50 report_08-22-07_13_06 report_08-22-07_14_40 report_11-01-07_11_27 report_11-01-07_11_38
FAST HELLO: report_08-21-07_14_32 report_08-21-07_17_39 report_09-10-07_11_31 report_09-10-07_11_43 report_09-10-07_11_54 report_09-10-07_12_05 report_09-10-07_12_15 report_09-10-07_14_50
medium_dense: report_08-17-07_13_41 report_08-17-07_13_54 report_08-17-07_14_08 report_08-17-07_14_20 report_08-17-07_14_32 report_08-17-07_14_44 report_09-16-07_20_01 report_09-16-07_20_14 report_09-16-07_20_40

INTENTIONALLY LEFT BLANK

Appendix C. The ParseIDSdata Class

```
#!/usr/bin/env python
#=========================================================
'''Author: bnguyen@arl.army.mil - Version 0.8 Thu 21 Feb 08'''

import  os, sys
import  glob
from    webbrowser import open_new

#=========================================================
# Predefined keywords in the input files:
#=========================================================
KEY_PREBANDWIDTH_COMSUMPTION  = 'PreBANDWIDTH COMSUMPTION'
KEY_BANDWIDTH_COMSUMPTION     = 'BANDWIDTH COMSUMPTION'
KEY_CALCULATIONS_LATENCY      = 'CALCULATIONS LATENCY'
KEY_META                      = 'Metadata'

#The metadata block has five fields:
KEY_NODE_DENSITY= 'NODE_DENSITY'
KEY_DATE_TIME   = 'DATE & TIME'
KEY_ATTACK_TYPE = 'Attack Type'
KEY_ATTACK_NODE = 'Attacker Node'
KEY_DETECTOR    = 'DETECTOR'
METADATA_KEYS   = ( KEY_NODE_DENSITY, KEY_DATE_TIME,                KEY_ATTACK_TYPE,
KEY_ATTACK_NODE, KEY_DETECTOR )

#=========================================================
DEFAULT_DATA_DIR = 'records'
DEFAULT_FN_PREFIX= 'report_'
#=========================================================

KEY_NAME    = 'Name'
KEY_VAL_SEP = '='
BLOCK_START = '{'
BLOCK_END   = '}'
COMMENT_CHAR= '%'

#=========================================================
def print_error(m): print >> sys.stderr, '***Error:', m
def print_dict(d):
   fmt='%%%ds : %%s' % max([len(x) for x in d.keys()])
   for k, v in d.iteritems():
      if type(v) == type( dict()):
         print 'Key = <%s>' % k
         print_dict(v)
      else:
         print fmt % (k, v)
   print

#=========================================================
```

```
class ParseIDSdata:
    def __init__(self, datdir=DEFAULT_DATA_DIR,
                                          prefix=DEFAULT_FN_PREFIX):
        #preparation
        data_list    = self.read_data(datdir, prefix)
        metalist, nameset, idsinfo = self.extract_info(data_list)

        attack_types = self.split_attack_types(data_list, idsinfo)
        #preservation
        self._metadata_list = metalist
        self._name_set      = nameset
        self._idsinfo       = idsinfo
        self._data_list     = data_list
        self._attack_types  = attack_types

        self.create_index_file(datdir, prefix)

    #-----------------------------------------------------
    #-----------------------------------------------------

    # getters() -- returning various data types ...
    def get_datalist(self):
        '''return data[i]=dict[k]'''
        return self._data_list

    def get_metadata_list(self):
        '''return list(dict[k])'''
        return self._metadata_list

    def get_name_set(self):
        '''return set([block-data names])'''
        return self._name_set

    def get_idsinfo(self):
        '''return dict[metadata-key]=set(values)'''
        return self._idsinfo

    def get_attack_types(self):
        '''return dict[attack-type]=list(data)'''
        return self._attack_types

    def extract_data(self, attack_type, density_type, detector_type):
        '''extract and return a list of data that meet the
           specifications (attack, density, detector)
        '''
        results =  list()
        for input_data in self.get_datalist():
            metadata = input_data.get(KEY_META)
            if metadata.get(KEY_ATTACK_TYPE) != attack_type:
                continue
            if metadata.get(KEY_NODE_DENSITY) != density_type:
                continue
            if metadata.get(KEY_DETECTOR) != detector_type:
                continue
            results.append(input_data)
```

```
    return results

def get_data_name_sets(self):
    ''' return list(set(k)), k is the name in the metadata block'''
    info = self.get_idsinfo()
    return (info.get(KEY_NODE_DENSITY),
            info.get(KEY_DATE_TIME),
            info.get(KEY_ATTACK_TYPE),
            info.get(KEY_ATTACK_NODE),
            info.get(KEY_DETECTOR))

    # ======================================================
    # the above statement = "get_idsinfo().values()"
    # The former is ordered, the latter is not.
    # ======================================================



# printers()/debuggers() ---
def print_attack_types(self):
    for attack, data in self.get_attack_types().iteritems():
        print 'AttackType (%s): list(dict[k]) (%d items)' % (attack, len(data))
def print_metadata(self):
    for metadata in self.get_metadata_list():
        print_dict(metadata)
def print_idsinfo(self):
        print_dict(self.get_idsinfo())
def print_name_set(self):
    for name in self.get_name_ set():
        print 'Unique Block Name:', name

#-----------------------------------------------------
# preparers -
#-----------------------------------------------------
def extract_info(self, data_list):
    metalist=  list()
    nameset =  set()
    idsinfo =  dict()    # idsinfo[k]= set()
    for mkey in METADATA_KEYS:
        idsinfo[mkey] =  set()
    for data in data_list:
        metadata = data.get(KEY_META)
        for k, v in metadata.iteritems():
            if k not in idsinfo:
                idsinfo[k]= set()
            idsinfo[k].add(v)
        metalist.append(metadata)
        nameset = nameset.union(data.keys())
    return metalist, nameset, idsinfo

def split_attack_types(self, data_list, idsinfo):
    '''requires: (1) a list of attack types + (2) list of data.'''
    results =  dict()
    for attack in idsinfo.get(KEY_ATTACK_TYPE):
        if attack not in results:
            results[attack]= list()
```

```
            for data in data_list:
                metadata = data.get(KEY_META)
                if attack == metadata.get(KEY_ATTACK_TYPE):
                    results[attack].append(data)
        return results

    def create_index_file(self, dirname, prefix, fname='index.htm'):
        '''create an index file organized according to the information
        described in the metadata block of each data file.
        '''
        #-----------------------------
        def html_hdrs(d):
            title = 'IDS Data Files in [%s] - bnguyen@arl.army.mil' % d
            return '<html><head><title>%s</title></head><body>' % title
        def html_tail():    return '</body></html>'
        #-----------------------------

        fd  = open(os.path.join(dirname, 'index.htm'), 'w')
        print >> fd, html_hdrs(dirname)

        results =  dict()
        for metadata in self.get_metadata_list():
            for k, v in metadata.iteritems():
                if k == KEY_DATE_TIME:
                    continue
                if v not in results:
                    results[v]= list()

                if 'win' in sys.platform: #':'-->'_'
                    results[v].append(metadata.get(KEY_DATE_TIME).replace(':', '_'))
                else:
                    results[v].append(metadata.get(KEY_DATE_TIME))

        for k, v in results.iteritems():
            print >> fd, '<P><B>%s:</B>' % (k)
            for suffix in v:
                name = prefix+suffix
                print >> fd, '<A HREF="%s">%s</A> ' % (name, name)
        print >> fd, html_tail()
        fd.close()

    #-------------------------------------------------
    #
    #-------------------------------------------------

    def read_data(self, datdir, prefix):
        def filter_data(fn):
            try:
                return [ s for s in [ s.strip() for s in open(fn).readlines() ] if len(s) and not
s.startswith(COMMENT_CHAR) ]
            except IOError, e:
                print_error(e)
                return  list()

        #----------------
        def parse_input(datalist):
```

```
        #-----------------------------------------
        def list2dict(s):
            return dict([(x[0].strip(), x[1].strip()) for x in
                        [x.split(KEY_VAL_SEP) for x in s if KEY_VAL_SEP in x ]])
        #-----------------------------------------

        results  =  dict()

                    blkstart = datalist.index(BLOCK_START)          results[KEY_META] = list2dict(
                    datalist[:blkstart] )
        if KEY_DETECTOR in results[KEY_META]: # Sparta -> SPARTA
            results[KEY_META][KEY_DETECTOR] =
                                        results[KEY_META].get(KEY_DETECTOR).upper()

        blkend  = datalist.index(BLOCK_END)
        while True:
            block = list2dict( datalist[ blkstart+1:blkend ] )
            results[block.get(KEY_NAME)] = block
            try:
                blkstart = datalist.index(BLOCK_START, blkend+1)
                blkend   = datalist.index(BLOCK_END, blkstart)
            except ValueError:
                break   # end of data.

        return results

    #----------------
    # read_data() ---
    #----------------

    if not os.path.isdir(datdir):
        raise SystemExit('<%s> is not a directory or does not exist' % (datdir))

    results =  list()

    for fn in glob.glob(os.path.join(datdir, prefix + '*')):
        input_data = filter_data(fn)
        if not input_data: continue
        results.append( parse_input(input_data) )

    return results

#============================================================
#
#============================================================
```

No. of Copies	Organization
1 (ELECT COPY)	ADMNSTR DEFNS TECHL INFO CTR ATTN DTIC OCP 8725 JOHN J KINGMAN RD STE 0944 FT BELVOIR VA 22060-6218
1	DARPA ATTN IXO S WELBY 3701 N FAIRFAX DR ARLINGTON VA 22203-1714
1	OFC OF THE SECY OF DEFNS ATTN ODDRE (R&AT) THE PENTAGON WASHINGTON DC 20301-3080
1	US ARMY RSRCH DEV AND ENGRG CMND ARMAMENT RSRCH DEV AND ENGRG CTR ARMAMENT ENGRG AND TECHNLGY CTR ATTN AMSRD AAR AEF T J MATTS BLDG 305 ABERDEEN PROVING GROUND MD 21005-5001
1	US ARMY TRADOC BATTLE LAB INTEGRATION & TECHL DIRCTRT ATTN ATCD B 10 WHISTLER LANE FT MONROE VA 23651-5850
1	US ARMY INFO SYS ENGRG CMND ATTN AMSEL IE TD F JENIA FT HUACHUCA AZ 85613-5300
1	COMMANDER US ARMY RDECOM ATTN AMSRD AMR W C MCCORKLE 5400 FOWLER RD REDSTONE ARSENAL AL 35898-5000
1	US ARMY RSRCH LAB ATTN AMSRD ARL CI OK TP TECHL LIB T LANDFRIED BLDG 4600 ABERDEEN PROVING GROUND MD 21005-5066
1	US GOVERNMENT PRINT OFF DEPOSITORY RECEIVING SECTION ATTN MAIL STOP IDAD J TATE 732 NORTH CAPITOL ST NW WASHINGTON DC 20402
1	DIRECTOR US ARMY RSRCH LAB ATTN AMSRD ARL RO E W D BACH PO BOX 12211 RESEARCH TRIANGLE PARK NC 27709
7	US ARMY RSRCH LAB ATTN AMSRD ARL CI NT B NGUYEN ATTN AMSRD ARL CI NT B RIVERA ATTN AMSRD ARL CI NT N IVANIC ATTN AMSRD ARL CI NT R HARDY ATTN AMSRD ARL CI OK T TECHL PUB ATTN AMSRD ARL CI OK T TECHL LIB ATTN IMNE ALC IMS MAIL & RECORDS MGMT ADELPHI MD 20783-1197
Total	17 (16 HC and 1 ELECT)